runviewer

Release 3.2.4

labscript suite contributors

Aug 03, 2025

DOCUMENTATION

Visualizes hardware-timed experiment instructions.

CHAPTER

ONE

INTRODUCTION

Runviewer is used for viewing, graphically, the expected changes in each output across one or more shots, and is shown in Fig. 1.1. Its use is optional, but can be extremely useful for debugging the behaviour of experiment logic. The output traces are generated directly from the set of hardware instructions stored in a given hdf5 file. This provides a faithful representation of what the hardware will actually do. In effect, runviewer provides a low level representation of the experiment, which complements the high level representation provided by the experiment logic written using the labscript API. As such, runviewer traces provide a way to view the quantisation of outputs,[2] which can be seen in the `central_Bq` and `central_bias_z_coil` channels in Fig. 1.1. You can also view the pseudoclock outputs. The `pulseblaster_0_ni_clock` and `pulseblaster_0_novatech_clock` channels demonstrate the independent clocking of devices from a single PulseBlaster pseudoclock. Similarly, `pulseblaster_1_clock` shows an entirely independent secondary pseudoclock.

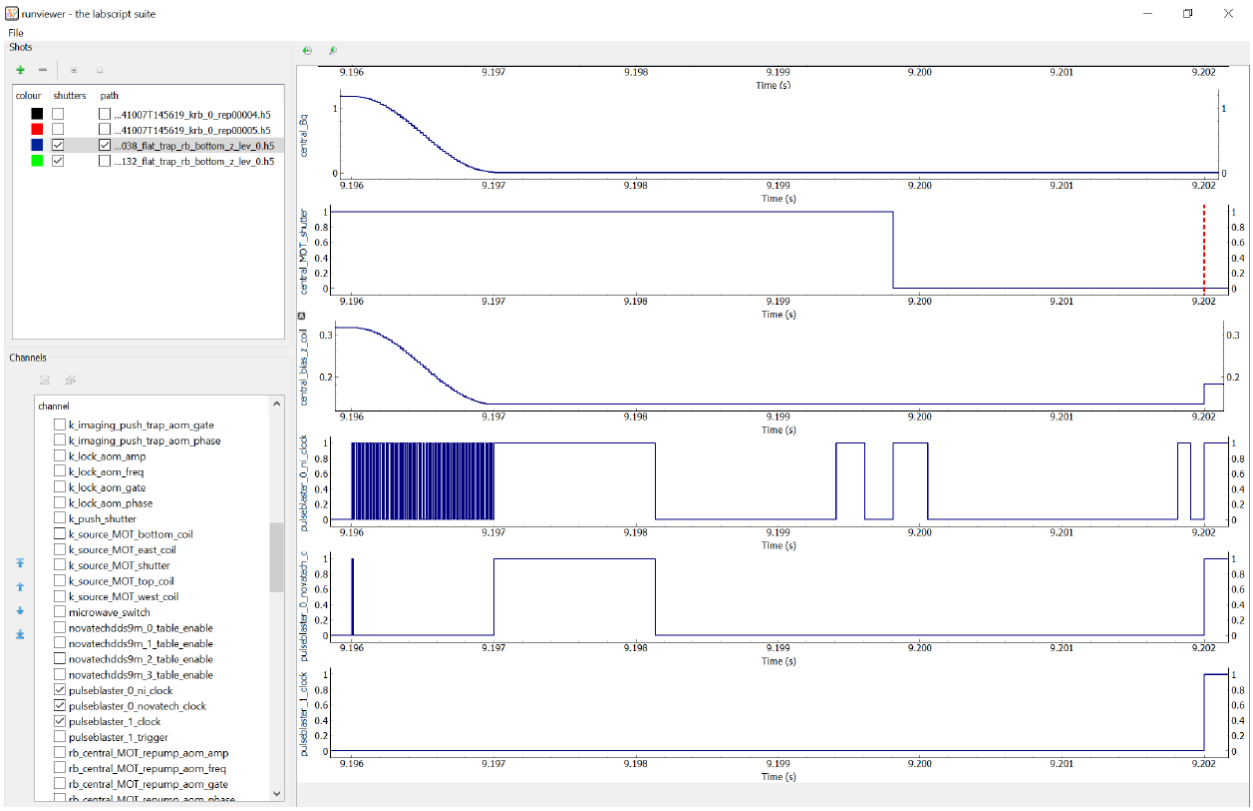

Fig. 1.1: An example of the runviewer interface.

[2] While this is always true in time, the output values may not be correctly quantised if the labscript device implementation does not quantise the output values correctly and instead relies on BLACS, the device programming API or the device firmware, to correctly quantise the output values.

CHAPTER

TWO

USAGE

While the textual based interface of labscript is ideal for defining experiment logic in 'high level' terms, there can often be a discrepancy between what was intended and what was actually commanded. This is particularly prevalent in situations where more complex control flow features (such as while loops or parameterised function) are used as this increases the abstraction between the language used to command the output and the actions of the output. Runviewer exists to bridge this gap, allowing us to maintain the benefits of textual control of the experiment without losing the benefits of the graphical representation of the experiment logic. Runviewer achieves this by producing a series of plots containing the output state for each channel. These plots are 'reverse-engineered' from the hardware instructions stored in the hdf5 file, and are thus a faithful representation of what each output channel should do during an experiment (provided of course that the reverse engineering code is accurate). There are thus several uses for runviewer. The most important is the ability to graphically observe the experiment logic. This allows a user to easily observe experiment features such as the shape of complex ramps or the synchronisation between events on different channels. Runviewer also supports simultaneous display of traces from multiple shots providing, for example, a means to see how an output trace changes when a global variable is adjusted. Finally, comparisons between expected output in runviewer, and observed output on an oscilloscope can make debugging hardware problems quicker.

2.1 Generating output traces

Runviewer generates the displayed output traces by processing the hardware instructions stored in the hdf5 shot file. We specifically reconstruct the output from the lowest level description of the experiment logic in order to accurately represent what the output hardware will do during an experiment. In order to support a diverse range of hardware, part of the reconstruction process is handled by device specific code that must be written by a developer when adding support for a new device. This device specific code simulates how the device processes hardware instructions and updates output states. It is discussed in more detail in labscript-devices, so for the purposes of this section we'll only cover generally what such code should do. The reconstruction algorithm is then as follows:

1. The master pseudoclock device is identified from the HDF5 file.
2. We import the device specific runviewer class for the master pseudoclock and request that it generate the traces for its outputs. As this is the master pseudoclock, we instruct the device specific code that there is not anything controlling the timing of this device (the need to do this will become apparent in a later step).
3. The device specific code generates a set of output traces (as it sees fit) and returns these traces to runviewer by calling a provided runviewer method, indicating that these traces should be available for display. This allows the device to produce as many traces as it likes, without limitation by the runviewer architecture. This is critical, as it removes the need for runviewer to support specific output types. Instead, this support is baked-in to the device specific code, which should already be aware of the output capabilities of the device.

   If timing information was provided by runviewer (which is the case for all devices except the master pseudoclock, see step 5 below), then it is used by the device code to generate the correct timing of the output traces. For example, a Novatech DDS9m only stores a table of output state changes, so the timing information of the parent ClockLine is needed. Similarly, the timing of a secondary pseudoclock is dependent on state changes to the parent Trigger line.

4. The device specific code then returns, to runviewer, a dictionary of traces for any ClockLines or Triggers assigned to digital outputs of the device (which may or may not have already been provided to runviewer for display in the previous step).

5. Runviewer iterates over this dictionary, and finds all devices that are children of each ClockLine or Trigger. For each device, the device specific code is imported and called as in step 2, except that this time we provide the device specific code with the trace for the ClockLine or Trigger so that it can generate output traces with the correct timing. The device specific code then follows step 3 and runviewer repeats steps 3 to 5 recursively until all devices have been processed.

2.2 The graphical interface

The graphical interface of runviewer comprises 3 sections (see Fig. 2.1). The first section manages the loading of shots into runviewer. Here you can enable (or disable) shots for plotting, choose the plot colour, and choose whether markers for shutter open and close times should be displayed. The second section manages the channels that are to be plotted. These channels can be reordered using the controls to the left, which then affects the order in which the plots appear. The list displays the union of all channels from shots that are currently enabled or have been previously enabled. This ensures runviewer remembers selected channels, even if they do not exist in the current shot, removing the need for a user to constantly re-enable channels when switching between different types of experiments. The configuration of enabled channels can also be saved and loaded from the 'File' menu, which is a useful aid when switching between regularly-used experiments.

The third section comprises the plotting region. We use the Python plotting library pyqtgraph to generate the plots. This choice was primarily made due to the performance of pyqtgraph, which is significantly faster than other common Python plotting libraries such as matplotlib.[3] The user can pan and zoom the plots produced by pyqtgraph using the mouse (by holding left or right mouse button respectively while moving the mouse). The time axes of each plot are linked together so that multiple output traces can be easily compared to each other. Two buttons are then provided at the top of the interface for resetting the axes to the default scaling.

As discussed previously, the output traces are generated directly from the hardware instructions. This creates two problems: information about the timing of certain events may not be contained within the hardware instructions, and the output trace may contain too many data points to plot efficiently (even when using pyqtgraph). The first problem we solve by plotting vertical markers at points of interest. For example, the Shutter class automatically accounts for the open and close delay of a shutter. The output trace thus only captures the time at which the digital output goes high or low and does not capture when the shutter will be open or closed. Runviewer reverse engineers these missing times from metadata stored within the hdf5 so that they can be plotted as markers of interest (see Fig. 2.2).

The second problem is solved by dynamically resampling the output traces depending on the zoom level of the x-axis of the plots. I wrote a feature-preserving algorithm for this purpose to avoid the many down-sampling algorithms that miss features faster than the sampling rate. This ensures that zoomed out plots accurately represent the trace, even when resampled. The algorithm starts by creating an output array of points that is 3 times the maximum width, in pixels, that the plot is expected to be displayed at. We fill every third data point in the output array using 'nearest neighbour on the left' interpolation, using only the section of the output trace that is currently visible. We then fill the other two data points with the highest and lowest value between the first data point and the 4th data point (which will also be determined using 'nearest neighbour on the left' interpolation). These two data point are placed in the order in which they appear, the reason for which will become clear shortly. This is repeated until the output array is full. The output array is then passed to pyqtgraph for plotting. Fast features thus exist in three data points of the array, which pyqtgraph correctly plots in one pixel as a vertical line. This is similar to the way digital oscilloscopes display acquired signals.

Despite our optimisation efforts, resampling still takes a significant period of time, particularly if there are many plots displayed. We thus perform the resampling in a thread in order to keep the GUI responsive. However, because the

[2] S. P. Johnstone, A. J. Groszek, P. T. Starkey, C. J. Billington, T. P. Simula, and K. Helmerson. *Evolution of large-scale flow from turbulence in a two-dimensional superfluid* Science **364**, 1267 (2019) https://doi.org/10.1126/science.aat5793

[3] We typically use matplotlib in the labscript suite as it is a widely known package with an almost identical syntax to MATLAB. This means that many users are already familiar with the syntax needed to create plots. As the user is not required to write or modify the code that generates the plots in runviewer, this benefit was not applicable and so it was worth using pyqtgraph for the increased performance.

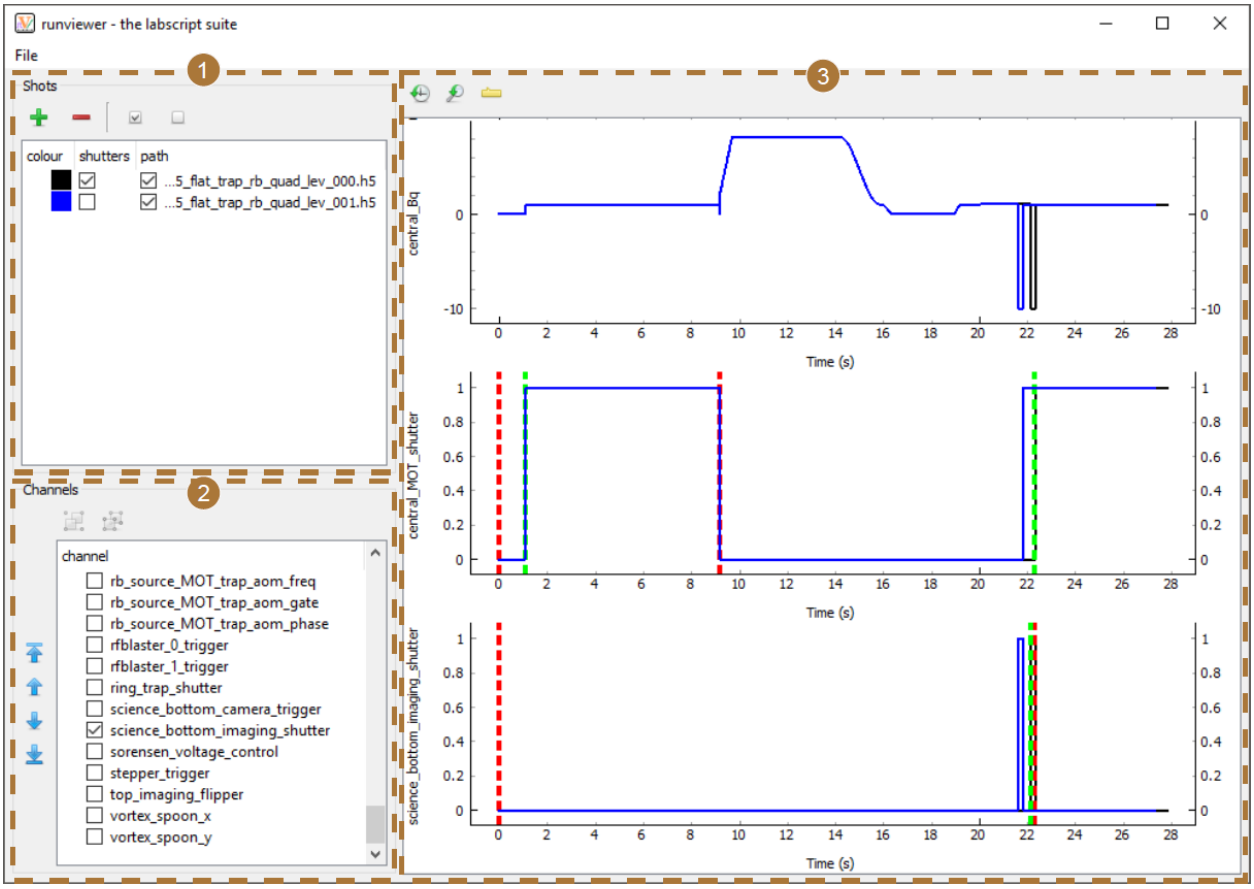

Fig. 2.1: The runviewer interface consists of 3 main sections. (1) Controls for loading shots, selecting the colour of traces, selecting whether shutter open/close markers are to be shown, and whether the output traces from this shot should be shown. Note that we have only enabled shutter markers for one of the two shots loaded (the black trace). (2) A reorderable list of channels contained within the loaded shots. The order here determines the order of plots in (3). Only enabled channels will be displayed in (3). (3) Plots of the output traces for the selected channels in the selected shots. Here we show data from 2 shots of a real experiment sequence from the Monash lab used to study vortex clustering dynamics[Page 6, 2]. The two shots loaded demonstrate how you can observe differences in output between shots in a sequence (in this case due to varying the time between stirring and imaging the vortex clusters). In this figure we display the entire length of the trace, which makes it difficult to distinguish between the shutter open/close events (red and green dashed, vertical lines) and the digital output trace. The discrepancy between these events becomes more apparent when zooming in (see Fig. 2.2).

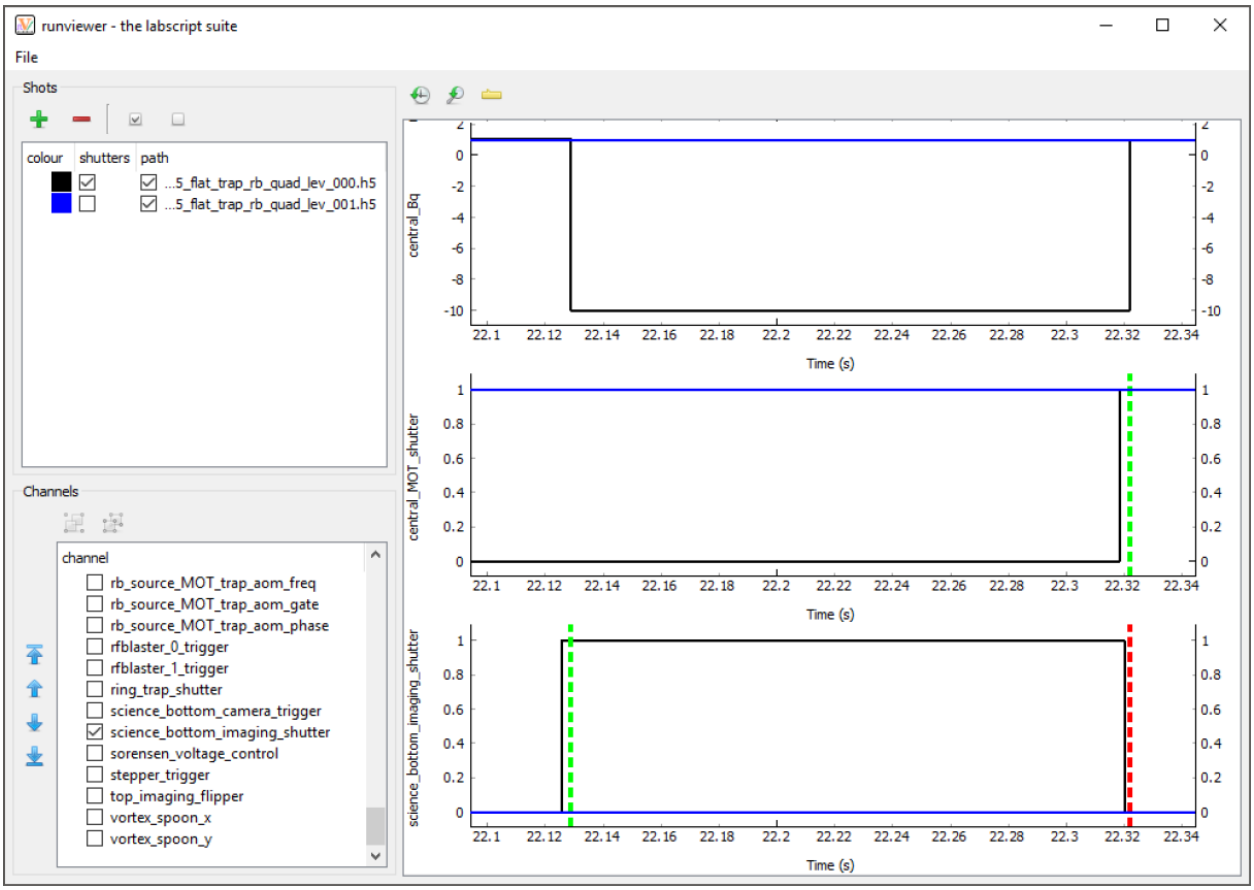

Fig. 2.2: Here we show the same traces as in Fig. 2.1, but zoomed just after the 22 s mark. We can now clearly see the difference between the change in digital state (black trace) used to open and close the shutter, and the time at which the shutter was actually commanded to open and close (green and red dashed, vertical lines respectively). In this case, the shutter (open, close) delay was specified in the labscript file as (3.11, 2.19) ms for the central_MOT_shutter and (3.16, 1.74) ms for the science_bottom_imaging_shutter.

resampled data has more points than can be displayed, and these points are in the correct order, zooming in still immediately shows a reasonable approximation of the trace while the user waits for the resampling to complete in the background.

CHAPTER

THREE

API REFERENCE

class runviewer.__main__.**ColourDelegate**(*view*, **args*, ***kwargs*)

Bases: QItemDelegate

createEditor(*self*, *parent: QWidget | None*, *option: QStyleOptionViewItem*, *index: QModelIndex*) → QWidget | None

get_next_colour()

setEditorData(*self*, *editor: QWidget | None*, *index: QModelIndex*)

setModelData(*self*, *editor: QWidget | None*, *model: QAbstractItemModel | None*, *index: QModelIndex*)

updateEditorGeometry(*self*, *editor: QWidget | None*, *option: QStyleOptionViewItem*, *index: QModelIndex*)

class runviewer.__main__.**RunViewer**(*exp_config*)

Bases: object

add_shutter_markers(*shot*, *channel*, *shutters_checked*)

channel_checked_and_enabled(*channel*)

create_plot(*channel*, *ticked_shots*)

get_selected_shots_and_colours()

load_shot(*filepath*)

mouseMovedEvent(*position*, *ui*, *name*)

on_add_shot()

on_load_channel_config()

on_remove_shots(*confirm=True*)

on_save_channel_config()

on_shot_selection_changed(*item*)

on_toggle_shutter(*checked*, *current_shot*)

on_x_axis_reset()

on_x_range_changed(**args*)

on_y_axes_reset()

resample(*data_x*, *data_y*, *xmin*, *xmax*, *stop_time*, *num_pixels*)

This is a function for downsampling the data before plotting it. Unlike using nearest neighbour interpolation, this method preserves the features of the plot. It chooses what value to use based on what values within a region are most different from the values it's already chosen. This way, spikes of a short duration won't just be skipped over as they would with any sort of interpolation.

update_channels_treeview()

update_plot_positions()

update_plots()

update_ticks_of_selected_shots(*state*)

class runviewer.__main__.**RunviewerServer**(**args*, ***kwargs*)

Bases: ZMQServer

handler(*h5_filepath*)

class runviewer.__main__.**ScaleHandler**(*input_times*, *target_positions*, *stop_time*)

Bases: object

class runviewer.__main__.**Shot**(*path*)

Bases: object

add_shutter_times(*shutters*)

add_trace(*name*, *trace*, *parent_device_name*, *connection*)

property channels

clear_cache()

delete_cache()

property markers

scaled_times(*channel*)

property shutter_times

property traces

class runviewer.__main__.**TempShot**(*i*)

Bases: *Shot*

property channels

get_traces()

runviewer.__main__.**format_time**(*input_sec*)

runviewer.__main__.**int_to_enum**(*enum_list*, *value*)

stupid hack to work around the fact that PySide screws with the type of a variable when it goes into a model. Enums are converted to ints, which then can't be interpreted by QColor correctly (for example) unfortunately Qt doesn't provide a python list structure of enums, so you have to build the list yourself.

CHAPTER

FOUR

LABSCRIPT SUITE COMPONENTS

The *labscript suite* is modular by design, and is comprised of:

Table 4.1: Python libraries

labscript — Expressive composition of hardware-timed experiments
labscript-devices — Plugin architecture for controlling experiment hardware
labscript-utils — Shared modules used by the *labscript suite*

Table 4.2: Graphical applications

runmanager — Graphical and remote interface to parameterized experiments
blacs — Graphical interface to scientific instruments and experiment supervision
lyse — Online analysis of live experiment data
runviewer — Visualize hardware-timed experiment instructions

PYTHON MODULE INDEX

r

INDEX

T

U